Hypertracing: Tracing Through Virtualization Layers

Abderrahmane Benbachir, *Member, IEEE,* Michel Dagenais, *Senior Member, IEEE*

Abstract—Cloud computing enables on-demand access to remote computing resources. It provides dynamic scalability and elasticity with a low upfront cost. As the adoption of this computing model is rapidly growing, this increases the system complexity, since virtual machines (VMs) running on multiple virtualization layers become very difficult to monitor without interfering with their performance. In this paper, we present hypertracing, a novel method for tracing VMs by using various paravirtualization techniques, enabling efficient monitoring across virtualization boundaries. Hypertracing is a monitoring infrastructure that facilitates seamless trace sharing among host and guests. Our toolchain can detect latencies and their root causes within VMs, even for boot-up and shutdown sequences, whereas existing tools fail to handle these cases. We propose a new hypervisor optimization, for handling efficient nested paravirtualization, which allows hypertracing to be enabled in any nested environment without triggering VM exit multiplication. This is a significant improvement over current monitoring tools, with their large I/O overhead associated with activating monitoring within each virtualization layer.

Index Terms—Virtual Machine, Para-virtualization, KVM, Performance analysis, Tracing.

1 Introduction

Cloud computing has emerged as another paradigm in which a user or an organization can dynamically rent storage resources and remote computing facilities. It allows application providers to assign resources on-request, and to adjust the quantity of resources to fit the workload. Cloud computing benefits lie in its Pay-as-you-Go model; users simply pay for the used resources and can adaptively increment or decrement the capacity of the resources assigned to them [1].

In any case, an essential metric in this dynamic process is relative to the fact that, irrespective of cloud users being able to request more resources at any time, some delay may be needed for the procured VMs to become available. Cloud providers require time to select a suitable node for the VM instance in their data centers, for resources to be allocated (such as IP addresses) to the VM, as well as to copy or boot or even configure the entire OS image [2].

This dynamic process is unavoidable to ensure cloud elasticity. A long undesirable latency during this process may result in a degradation of elasticity responsiveness, which will immediately hurt the application performance.

Paravirtualization is the communication that happens between the hypervisor and the guest OS to enhance I/O efficiency and performance. This entails changing the OS kernel for the non-virtualizable instructions to be replaced with hypercalls that simply communicate with the virtualization layer hypervisor. The hypervisor, likewise, offers hypercall interfaces for related critical kernel activities like interrupt handling, timekeeping, and memory management [3].

- *The authors are with the Department of Computer and Software Engineering, Polytechnique Montreal, Montreal, Qubec, Canada. E-mail: {abderrahmane.benbachir, michel.dagenais}@polymtl.ca.*

Manuscript received March X, 2018; revised Month Y, 2018.

Boot-up and shutdown phases are critical stages that a system must undergo to initialize or release it resources. During these stages, many resources may not be available, such as storage, network or memory allocation. As a consequence, most debugging and tracing tools are either not available or with very limited capabilities. Therefore, those stages are particularly difficult to trace or debug.

An important challenge is to monitor guest systems while they go through these critical stages. Current monitoring tools were not conceived to operate in such conditions. To overcome those limitations, we propose paravirtualization solutions to be integrated into current monitoring tools.

In this paper, we propose **hypertracing**, a guest-host collaboration and communication design for monitoring purposes. In particular, we developed trace sharing techniques that enable accurate latency detection, which is not addressed by existing monitoring tools. We propose an approach based on different paravirtualization mechanisms, which is an effective and efficient way to communicate directly with the host, even from nested layers. The proposed method consists in offloading and merging guests and host traces. Depending on which communication channels are used, trace fusion may need synchronization to insure that events are stored in chronological order.

Our main contributions in this paper are: **First**, we propose an hypercall interface as a new communication channel for trace sharing between host and guests, comparing this with shared-memory-based channels. The hypercall channel allows us to debug VM performance issues, even during sensitive phases such as boot-up and shutdown. **Secondly** we propose a technique to enable guest virtualization awareness, without accessing the host. **Thirdly** we submitted many kernel patches to the Linux community to perform boot-level tracing and enable function tracing during early boot-up. **Fourthly** we

developed a KVM optimization patch for handling nested paravirtualization, which prevents exit multiplication and reduces CPU cache pollution when performing hypertracing from nested layers. **Lastly**, we implemented VM analysis views which improve the overall performance investigation.

The rest of this paper is structured as follows: Section 2 presents a summary of existing paravirtualization approaches used for VM monitoring. Section 3 states the problem addressed by this paper. Section 5 introduces a comparative study of existing inter-VM communication channels. In section 6 we present the design and architecture of our hypertracing techniques. Section 7 outlines some performance challenges of using hypertracing from nested environments and how we addressed them. Section 8 shows some representative use cases and their analysis results, followed by the overhead analysis in section 9. Finally, Section 10 concludes the paper with future directions.

2 Related Work

Various monitoring tools embraced paravirtualization to reduce I/O performance issues while tracing VMs. In this section, we summarize most of the previous studies related to our work, grouped into the following categories.

2.1 Hypercall based

Khen at al. [4] presents LgDb, a framework tool that uses Lguest to provide kernel development and testing, code profiling and code coverage. It enables running within a virtual environment kernel modules under study. The key behind LgDB is similar to traditional debuggers, as the framework will cause a context switch from the guest to the hypervisor mode, in order to stop guest execution when code flow reaches a specific point. Two approaches were proposed for this matter. First, the hypercall-based approach, which requires manual instrumentation of the inspected kernel code to set breakpoints. Secondly, the debugger-based approach which does not require any code modification, uses the kernel debugger (KGDB) to enable breakpoints over a virtio serial port.

Stolfa at al. [5] propose Dtrace-virt, an extension on top of the DTrace tracer in FreeBSD. It is a monitoring tool tailored for tracing untrusted VMs. DTrace-virt was designed as an Intrusion Detection System (IDS) and malware analysis tools; this tool has been designed to detect malicious code injections by sending aggregated trace event data to the host hypervisor, using hypercalls as a fast trap mechanism, instead of using networking protocols (traditionally used by many distributed tracing tools) which induce a larger overhead in a virtualized environment. The authors did not provide any use case to evaluate the usefulness of such a monitoring approach for security purposes; their work only discusses the infrastructure design and implementation challenges of their approach.

Gebai at al. [6] proposed a multi-level trace analysis which retrieves the preemption state for vCPUs by using merged traces from the host kernel and each VM. His work was extended by Biancheri [7] for nested VMs analysis. Both study the root cause of preemption by recovering the execution flow of a specific process, whether it is a host thread, VM or nested VM. They proposed an approach to resolve clock drift issues using paravirtualization with periodic hypercalls as synchronization events. Their approach adds about a 7% constant overhead to each VM, even if they are idle.

2.2 Page sharing based

Yunomae [8] implemented virtio-trace, a low-overhead monitoring system for collecting guests kernel trace events directly from the host side over virtio-serial, in order to prevent using the network stack layers while sending traces to host. They enable an agent inside the guest to perform splicing operations on Ftrace [9] ring buffers, when tracing is enabled. This mechanism provides efficient data transfers, without copying memory to the QEMU virtio-ring, which is then consumed directly by the host.

Jin at al. [10], developed a paravirtualized tool named Xenrelay. This tool is an unified one-way inter-VM transmission engine mechanism, very efficient for transferring frequently small trace data from the guest domain to the privileged domain. Xenrelay has the ability to relay data without lock or notification. This mechanism was implemented using a producer-consumer circular buffer with a mapping mechanism to avoid using synchronization. It helped to optimize Xen virtualization performance by improving the tracing and analysis virtualization issues.

2.3 Hypervisor monitoring

Nemati [11], [12] proposed a new approach to trace guests virtual CPU states, in any virtualization layer, tracing only the host without accessing VMs. Sharma [13], used a hardware tracing monitoring-based approach to generate hypervisor metrics and exposing virtualization overhead. This method involves continuous host access, which may not be possible in some cases. Our work is unique in the sense that it can be used without the need to access the host. We use paravirtualization to enable effective host-guest communication and collaboration. This enables virtualization-awareness from the guest perspective, and enables measuring the virtualization overhead from the guest.

Other studies [14], [15] attempt to measure the virtualization cost using performance benchmarks suites, in order to measure CPU, memory and I/O virtualization overhead. However, such an approach is not representative of real-world workloads.

Paravirtualization approaches are mostly used to avoid the heavy network path communication, using shared memory or hypercalls as fast communication paths in virtualized environments. To our knowledge, no previous study has used this mechanism to solve real world performance issues, for instance at bootup or shutdown when few operating system facilities are available; paravirtualization approaches were mostly used to speedup I/O operations.

3 Problem Statement and Definitions

In this section, we present challenging issues, grouped into the following categories.

3.1 Nested Layers

Various performance issues in virtual environments are caused by the isolation layer imposed by the hypervisor. VMs have the illusion of exclusive access to system hardware due to the virtualization layer. When a system has multiple layers of virtualization, it becomes very difficult to know what is happening. As a basic approach, we can enable monitoring in each layer. However, there is still the challenge to merge and synchronize the traces coming from multiple layers due to clock drift. Moreover, monitoring in all layers comes with overhead concerns, as most monitoring tools need to consume the recorded data in one way or another, either by storing them to disk (offline analysis), or using live monitoring to offload them through a network (online analysis). Both approaches introduce significant overhead related to I/O operations, especially when tracing is enabled within a nested environment. In a nested environment, the performance slowdown could reach a factor of 30 and more.

3.2 Crash Dump

QEMU has a panic device, supported by Libvirt since the first version. Using this device, the guest is capable of notifying the host when a guest panic occurs, through the process of sending a particular event directly to QEMU, which in turn will inform Libvirt.

Libvirt offers a mechanism that can store guest crash dumps automatically to a dedicated address on a host. Activating the panic device PVPanic for guests will enable a crash dump to be forwarded, without the need of using kdump [16] within the guest [17]. The major issue with this method is the amount of data generated. A basic crash dump size would be 128MB. If a system has 1TB of memory, then 192MB will be reserved, a 128MB of basic memory plus an additional 64MB. As a result, the host would need more storage resources to store all these crash dumps coming from different guests.

3.3 Boot Time Analysis

The Boot-up time is the amount of time it takes to boot a VM into a ready state. It is an important factor for VM allocation strategies, particularly while provisioning for peak load. Automated allocation strategies will request new VMs to match load demand. In this situation, the time to boot VMs is critical, as the system is waiting after the boot-up in order to be fully operational. Enabling monitoring during the boot-up phase is challenging, yet is often the only effective way to investigate boot-up latencies. Data generated during this phase cannot be stored on disk or offloaded through the network, because network and storage are only available once this phase is almost completed.

Another approach may be possible. If enough memory space is allocated at the beginning of boot-up, to prevent event loss, then events can be consumed after boot-up when storage and network are available. However, this approach requires a large memory space allocation (ring buffer). It also adds a significant time overhead because the monitoring tool uses memory frame allocation at initialization time [18], avoiding postponed page faults while tracing is enabled. This initialization, at the start of the boot process, directly impacts the boot time.

Kernel crashes may also happen while VMs are booting. Investigating such problems is a non-trivial task since most of the debugging tools are not available at that stage. Kernel developers typically resort to using printk for dumping messages on the serial console, with information such as call stacks and register values. Using printk introduces a very large overhead (system calls) and can affect the timing, particularly when the output is over a serial line [19], which does not help to solve complex performance issues.

3.4 Shutdown Analysis

Investigating the shutdown latency is another challenging task. One may question the interest of optimizing the shutdown latency. In many cases, latencies are not very important when turning off VMs during the under-provisioning stage. Nevertheless, VMs shutting down consume resources and may prevent a host from starting up more rapidly newer VMs. Shutting down existing VMs quickly is the fastest way to get back the necessary resources. Thus, any delay during the shutdown is directly impacting the provisioning of newer VMs. However, in some cases, the shutdown latency is much more critical. Indeed, in critical embedded systems, the shutdown latency is on the critical path for the case where your system needs to reboot after doing a major software upgrade. Any significant delay impacts the non availability phase during the upgrade.

When a system is shutting down, userland processes are being released, drivers being unloaded, and the network and disks are not available. Current monitoring tools are not designed to operate during this phase.

In this paper, we present how we use paravirtualization to design guest-host collaboration and address the above challenges. As we illustrate in section Use cases, the techniques resulting from our work will help to efficiently investigate system failures and hidden latencies within VMs, as well as enabling runtime perturbations detection due to CPU or disk sharing between colocated VMs.

4 Environment

For all experiments and benchmarks, we used a computer with a quad-core Intel(R) Core(TM) i7-6700K CPU running at 4.00GHz, 32 GB of DDR3 memory and a 256 GB SSD. The operating system is Ubuntu 16.10, running Linux 4.14 and LTTng 2.10. The frequency scaling governor of the CPU was always set at performance to prevent CPU scaling operations while performing the benchmarks. Benchmark results were computed using the average of one million iterations.

KVM and QEMU 2.8.0 were used as virtualization technology. Host and guests were configured with the same linux versions during all experiments. For nested environments, we enabled all nested optimizations such as the EPT-on-EPT feature.

5 Inter-VM Communication Channels Analysis

Virtualization technology provides security and isolation, with the ability to share system resources between VMs co-located on the same host. Furthermore, isolation is the main property that keeps the industry from shifting to container technology. However, when co-located VMs need to communicate with each other or with the host, it becomes a bottleneck.

Communication by means of TCP/IP requires a longer time since the transfer of data from the sender VM directly to the receiver host undergoes a long communication channel through a host hypervisor that incurs multiple switches between the root mode and user mode, as well as going across the entire network stack layers. This is inherently inefficient and causes performance degradation. Moreover, to allow efficient and fast communication, the guest OS and the hypervisor are capable of using an optimized I/O interface, rather than depending on TCP/IP. This mechanism is called **paravirtualization** [20].

In this section, we discuss different paravirtualization communication channel alternatives to TCP/IP, exposing design limitations and performance drawbacks of each one.

5.1 Page Sharing Channel

Instead of emulating hardware devices, Xen designed a new split driver I/O model, which provides efficient I/O virtualization and offers protection and isolation of the device interface. In this new model, drivers are split into frontend and backend, isolated from each other across Xen domains. These frontend and backend can only communicate asynchronously through the hypervisor, in order to satisfy the isolation, using shared-memory (shared I/O ring buffers) [21].

The shared I/O ring buffers are enabled by two main communication channels provided by Xen, the Grant Table and the Event mechanism. The event mechanism allow inter-VM asynchronous notification and signals, instead of polling. Events can be used to signal received/pending network data, or to indicate the completion of a disk request. The Grant Table is the fundamental core component for memory sharing. It offers mapping/transfer memory pages between frontend and backend drivers. Moreover, the grant mechanism provides an API, accessible directly from the OS kernel, to share guest pages efficiently and securely. It allows guest domains to revoke page access at any time.

The Xen page sharing mechanism is implemented by issuing synchronous calls to the hypervisor (hypercalls). If a guest domain wants to share a page, it invokes a hypercall that contains the target domain reference and the shared page address. The hypervisor will first validate the page ownership, then map the page into the address space of the target domain. Similarly, to revoke a shared page, the guest issues another hypercall, and this time the hypervisor unmaps and unpins the guest page from the target address space [22].

In early 2006, when KVM was merged into linux, virtio was introduced. It was based on Xen I/O paravirtualization, which uses a mechanism similar to the Grant Table. Afterwards, virtio evolved and converged into the default I/O infrastructure supported by different hypervisors.

Virtio is an efficient paravirtualization framework that provides a transport abstraction layer over hardware devices. It offers a common API to reduce code duplication in virtual device drivers. Such drivers enable efficient transport mechanisms for guest-host as well as inter-guest communication channels [23]. Similar to Xen, virtio device drivers are also split into the frontend which resides in the guest OS, while the backend can be either in user space, inside QEMU, or in kernel space as KVM modules (vhost). The communication between those drivers (guest and hypervisor) is organized as two separate circular buffers (virtqueue), in a producer-consumer fashion, as illustrated in Fig. 1. The avail-ring is for sending and the used-ring for receiving.

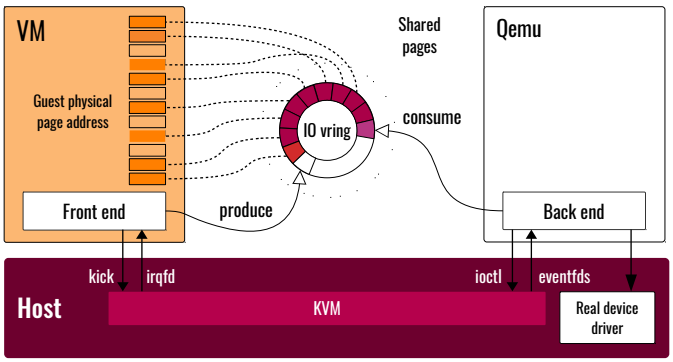

Fig. 1. Architecture of Page Sharing mechanism in virtio

A producer (guest driver) writes data into the virtqueue avail-ring in the form of scatter-gather (guest physical address pointer and length) lists, then notifies the host when buffers are available, using the kick call mechanism. The virtio kick operation can be performed using the *paravirt_ops* API infrastructure. This API can be either an input/output port access (*pio*) operation, a hypercall or other notification mechanism which depends on the hypervisor. The host can decide to disable guest notifications while processing buffers, as notification implies an expensive context switch of the guest. Virtio also supports batching, which improves the performance throughout. This feature can be enabled by adding multiple buffers to the virtqueue before calling the kick event [24]. When the consumer (hypervisor or QEMU) gets the notification, it reads the virtqueue avail-ring buffers. The consumed buffers will then be pushed to the used-ring.

A virtio device may choose to define as many virtqueues as needed. For instance, the virtio-net device has two virtqueues, the first one for packet transmission and the other one for packet reception.

Virtio-serial is an implementation of the serial bus on QEMU. This mechanism was used for monitoring purposes, such as retrieving metrics of the guest CPU, memory and disk usage [25]. As presented in section 2, virtio-serial was also used as a transport layer to implement virtio-trace. We decided to use virtio-serial as the reference implementation for the page sharing channel, when we compare it to other inter-VM communication mechanisms later in this section.

5.2 Memory Sharing Region Channel

Inter-Process Communication (IPC) is an operating system mechanism allowing different processes to collaborate and share data. A variety of IPC mechanisms are supported in most operating systems, including sockets, signals, pipes and shared memory. Depending on the system configuration, processes may be distributed across different machines, or across different data centers, which requires using remote procedure calls (RPCs) and streaming data over the network. On the other hand, processes may be located on the same machine and, in this case, it is more efficient to use a shared memory to increase the performance of concurrent processing.

With the recent advances in virtualization technology, the need for efficient Inter-VM communication is growing. A variety of inter-VM shared memory mechanisms were introduced in the Xen hypervisor using a page-flipping (Grant) mechanism. Nahanni and ZIVM are two shared memory mechanisms that were introduced in KVM. Unlike Xen, these mechanisms allow host-guest as well as guest-guest sharing, supporting POSIX and System V shared memory regions.

Developed by Cameron Macdonell, Nahanni is a zero-copy low-latency and high-bandwidth POSIX shared memory based communication channel. It enables hosts, guests, as well as inter-guests efficient transport mechanisms. Unlike a page sharing mechanism, it does not require any hypervisor or guest involvement while reading or writing to the shared region [26].

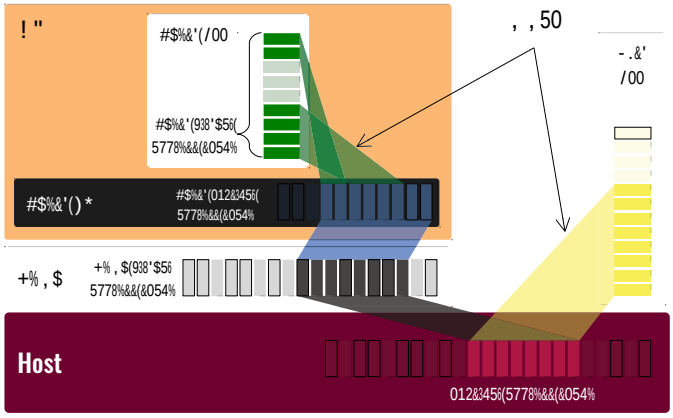

Fig. 2. Architecture of Memory Sharing Region

As illustrated in Fig. 2, Nahanni is implemented as a new virtual PCI device (ivshmem) in QEMU, which will be used to share the POSIX memory region, mapped into QEMU with the guest OS. Userspace applications inside the guest OS can access directly the shared memory region by opening the device using the *mmap* system call. This call will map the memory region into their address space, with zero-copy. Nahanni also supports a notification mechanism, in order to notify the host or other guests about data availability. It uses event file descriptors (eventfds) as an interrupt for signalling and synchronization mechanisms.

We decided to use Nahanni as the reference implementation for the memory sharing channel, after we compared it with other inter-VM communication mechanisms.

5.3 Hypercall Channel

With the introduction of Hardware-assisted virtualization in Intel-VT and AMD-V, all privileged instructions executed in user-mode (guest OS) should cause a trap into the root-mode (hypervisor). This transition incurs significant overhead and is called Virtual Machine Extensions (VMX). By using paravirtualization we can reduce this overhead, a guest OS source code will be modified to replace those VMX (privileged) instructions with a single VMX instruction called hypercall, which will invoke the hypervisor to perform heavy-weight work. The hypercall interface allows the request of a service from the hypervisor by performing a synchronous software trap into the host, which is similar to system calls. Fig. 3 illustrates the hypercall trap mechanism in more details.

In other words, the hypercall is a guest request to the host to perform a privileged operation, and it can be used as a guest to host communication mechanism as well.

The hypercall transport mechanism is performed using registers. As an illustration for x86 architecture, guests could use the default general purpose registers **ax**, **bx**, **cx**, **dx** and **si** to place data before emitting the hypercall. The ax register will be reserved for the hypercall number during the submission, then the hypervisor will use **ax** to place the returned value when switching back to user mode [27]. On the Intel 64-bits architecture, additional registers can be used such as **r8** to **r15**, coupled with a proper compiler register clobbering mechanism.

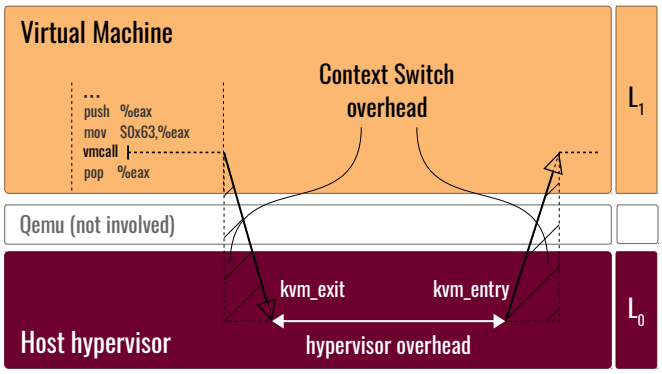

Fig. 3. Hypercall in a nutshell

A practical hypercall use case is available in the Linux project. A hypercall can be used by guests to wake up a virtual CPU (vCPU) from the halt (HLT) state. In case a given thread is waiting to access a resource, the thread would perform an active polling (spinlock) on a vCPU. Paravirtualization can be used in this case to save wasted cpu cycles, by executing the HLT instruction when a time interval threshold was elapsed. The execution of the HLT instruction causes a trap to the hypervisor, and then the vCPU will be put into sleep mode until an appropriate event is raised. When the resource is released by other threads, on other vCPUs, the thread can wake up the sleeping vCPU by the triggering KVM_HC_KICK_CPU hypercall [27].

5.4 Performance Comparisons

In this section, we present the performance comparison of three data transmission channels: hypercall, virtio-serial and Nahanni. We measure the bandwidth for streaming a fixed volume of data (1 GB), from guest (with single vCPU) to host repeatedly. Our bandwidth benchmark measures the time to send the data to the host, without measuring the overhead of consuming it. There are several ways for consuming the data, and in this section we only need to compare the transmission path between these channels, which is enough for a micro-benchmark. Later in section 9 we will compare these channels with a more realistic workload.

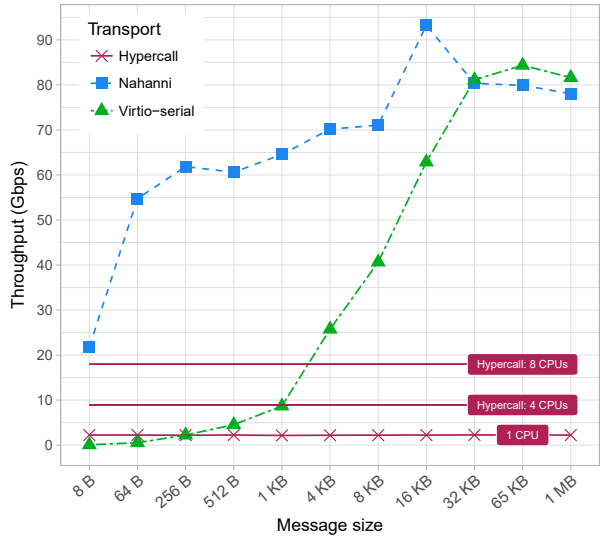

Fig. 4. Throughput versus message size for Inter-VM communication transport benchmarks

Fig. 4 shows the bandwidth measured with hypercall, virtio-serial and Nahanni while increasing the sending message size. The bandwidth increases for both virtio and Nahanni with larger message sizes. Then, both reach a stable throughput of 80 Gbps when message sizes are larger than 32 KB. Nahanni achieves higher throughput for smaller and larger message data than both the hypercall and virtio mechanisms. Virtio shows the worst bandwidth for sending smaller messages. This is because it requires a larger number of hypervisor exits to send the same amount of bytes.

The hypercall achieves a constant throughput, of about 2 Gbps, with a single CPU, and 17.5 Gbps with 8 CPUs. This is because the hypercall uses CPU registers for the transmission. Increasing the message size simply implies using more registers (if available). We couldn't use more than 8 CPUs due to resource limitations.

In summary, shared memory based approaches achieved a higher communication efficiency as compared to other channels.

To solve the challenging problems previously raised in section 3, we decided to explore in depth both hypercall and shared memory region channels, we have found that there is a lack of scientific contribution for both channels compared to the page sharing channel, which had enormous contributions on either virtio KVM or Xen due to their popularity. The next section presents the design of hypertracing techniques using both channels.

6 Hypertracing Design and Architecture

In this section, we will explain the design and architecture of hypertracing using both hypercall and memory sharing channels.

6.1 Hypertracing Through Hypercall

As explained in section 5, a hypercall is a guest synchronous request to the host to perform a privileged operation, but it can also be used for debugging, or for guest-host communication purposes. The heart of a hypercall is the *vmcall* VMX instruction, the main aim of which is to cause an instant trap into the host hypervisor. On the other hand, the transport mechanism of a hypercall can be performed solely by using guest registers. The approach of using the *vmcall* instruction alongside the registers requires saving these registers into the guest stack. When returning from the hypercall, the guest can then restore the registers state, from the stack, before continuing it execution. The *vmcall* instruction can also be used without touching guest registers. Instead, it can be used for the purpose of inspecting guest registers, for instance as a breakpoint mechanism.

6.1.1 *Debugging aspect*

The *vmcall* instruction has a behavior similar to debugger breakpoint mechanisms, which use the interrupt instruction (*INT 3*) to cause a userspace program to trap into the kernel space.

A practical example of debugging a guest kernel using *vmcall*, would be the function tracing. Adding the gcc *-pg* option while compiling the kernel will automatically add a mcount() call at the beginning of each function. Ftrace [9], the kernel tracer, relies on the gcc *-pg* option for its function tracing infrastructure. During boot-up, Ftrace will use live patching to convert all mcount() calls into *NOP* instructions. When tracing is enabled, Ftrace converts the *NOP* instructions into a callback routine that handles the tracing part. This mechanism is known as dynamic tracing.

To be able to debug a guest kernel from the host side, we can use dynamic tracing coupled with *vmcall*. This mechanism will convert dynamically the *NOP* instructions into a *vmcall* instruction. Each time a kernel function is called in the guest, a trap to the host hypervisor happens. Once in the host, it is possible to inspect the guest registers, stack, heap and memory directly from the host. This allows the extraction of guest data such as function address or function arguments. It is possible to use a pause/resume operation on a guest, when a specific hypercall was triggered. This feature will allow developers to manually inspect guests for advanced step-by-step investigations.

6.1.2 Tracing aspect

Debugging is a very useful technique, providing developers with a handful of features to investigate very complex problems in step-by-step mode. However, debugging in production cloud-based environments is not tolerable due to its huge overhead cost. Instead, practitioners use tracing tools to collect VM execution events in the most optimal way. Then, collected events would be asynchronously recorded on disk for offline analysis.

Another way of tracing virtual environments would be to offload guest events directly to the host. This mechanism requires using a transport channel, such as hypercall. When offloading guest events using hypercalls, there are three different offloading configurations to consider, discussed as follows:

One-to-One: 1-event to 1-hypercall This configuration restricts the data transmission to only one hypercall per event. Each guest event will cause a trap in the host hypervisor, using host timestamps when recording the guest event.

In the case of high event frequency, this configuration may dramatically impact the overall performance of the guest execution. However, no latency is added to traced events and, as a result, we have instant insight into what is happening inside guests, from a host perspective with aligned timing. This mechanism allows tracing many guests at the same time in live monitoring.

One-to-Many: 1-event to n-hypercalls This configuration may help the situation when a specific event needs more memory space than available (e.g. fixed number of registers) to send a payload. In this case, two or three hypercalls may be used for the transmission, resulting in an enormous performance impact on the guest execution. In this paper, we didn't find a need for this configuration. First, all traced event data payloads did fit between 1 to 4 registers. Secondly, a shared page can be shared between the host and guest in order to expand the hypercall payload maximum size.

Many-to-One: n-events to 1-hypercall The purpose of this configuration is mainly to reduce the overhead induced by a hypercall. The idea is simple, multiple hypercalls can be batched as a single hypercall. This is important to reduce the number of hypercalls being used for sending events. The more hypercalls are batched, the less overhead we get. We named this mechanism the **event batching** and **event compression**.

Event Batching When event batching is enabled, we save the payload and record the guest timestamp on each event. When the last event of the batch is encountered, we group the events data as much as possible to fit into the hypercall payload. Instead of sending a guest timestamp, we send the time delta of each event, which would be computed against the guest last event. The last event will trigger a hypercall and a timestamp will then be recorded in the host, which will be used to convert the batched events times delta into host timestamps. Storing the time delta requires only 32 bits, and enables batching events within a maximum interval of 4 seconds. This is sufficient for the groupings envisioned.

Even with a minimal configuration, event batching turns out to be very efficient. However, this mechanism would mostly work for periodic events that happen frequently. Moreover, finding the appropriate number of events to batch largely depends on the type of events being traced. The main drawback of batching is the latency added before an event becomes visible, for instance in live tracing. Another problematic scenario is upon a crash, the crucial last few events before the crash may be held up in a batch being assembled.

In order to fit as many events as possible in the payload, the batched events should be periodic, frequent, and from the same event type. These properties are already present in most operating system applications. Nonetheless, in this paper we focus on the Linux operating system.

Linux offers a robust tracing infrastructure that supports static and dynamic instrumentation. Static tracepoints have been manually inserted into different subsystems, allowing developers to better understand the kernel runtime issues. Scheduling, memory management, filesystems, device drivers, system calls and many more are important components in the kernel, and they are exercised frequently during the execution. These components are already instrumented, and they can efficiently be used with event batching since they occur periodically in the system. As an illustration, Fig. 5 provides more details on how batching scheduling events could be implemented. We have used the sched_switch event as an example in this figure.

Sched_switch is the linux trace event that shows the context switches between tasks. This event allows recording information such as the name, priority and tid of the previous and the next tasks. This is represented by seven fields as follows: prev_prio, prev_state, prev_tid, prev_comm, next_prio, next_tid and next_comm. In linux, the pid/tid maximum value is configurable, but pid has a default maximum value of 32768, which needs only 15 bits to be stored. The task priority and it state can be stored within 8 bits each. The task name length is 16 characters, and requires two registers to be stored.

If we use a one-to-one configuration, each sched_switch event would require 5 registers (64 bits architecture), which can easily fit into a hypercall payload. Batching sched_switch events is much easier than it looks. As has been noted previously, the sched_switch event contains data fields about the previous and next tasks. We notice that, when analyzing two consecutive sched_switch events, as shown in Fig. 5, the task "migration/1" which was included as the "next task" in the first sched_switch event, is also included in the second sched_switch event as the "previous task". Therefore, **event compression** can be enabled, omitting the repeated information, in order to batch more sched_switch events within the payload.

The effectiveness of any compression algorithm is evaluated by how much the input size is reduced, which depends on the input data. In our case, the situation is different. We have a predefined and fixed length of hypercall payload, and our goal is to fit in that payload as many events as possible, without losing any information.

With compression enabled, event batching will take into consideration only the "next task" fields, as the "previous task" fields can be retrieved from the previous event while

```
sched_switch { prev_prio = 0, prev_tid = 0, next_tid = 10, prev_state = 0, next_prio = 0, prev_comm = "swapper/1", next_comm = "migration/1" }
...
sched_switch { prev_prio = 0, prev_tid = 10, next_tid = 11, prev_state = 1, next_prio = 0, prev_comm = "migration/1", next_comm = "ksoftirqd/1" }
sched_switch { prev_prio = 0, prev_tid = 11, next_tid = 12, prev_state = 1, next_prio = 0, prev_comm = "ksoftirqd/1", next_comm = "kworker/1:0" }
sched_switch { prev_prio = 0, prev_tid = 12, next_tid = 0, prev_state = 1, next_prio = 0, prev_comm = "kworker/1:0", next_comm = "swapper/1" }
sched_switch { prev_prio = 0, prev_tid = 0, next_tid = 10, prev_state = 0, next_prio = 0, prev_comm = "swapper/1", next_comm = "migration/1" }
```

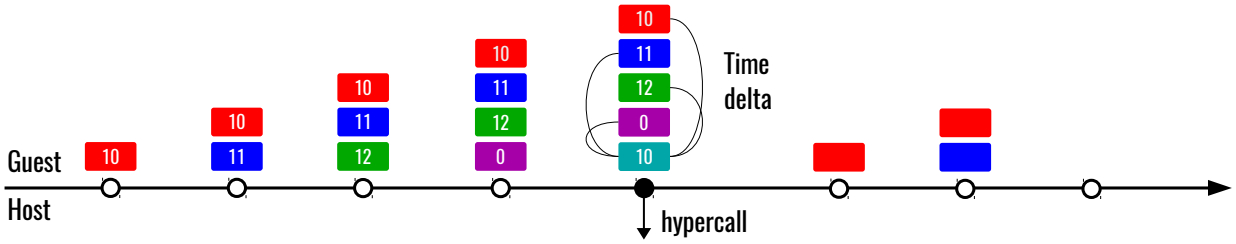

Fig. 5. Event batching of schedule switches events with compression enabled.

consuming these events from the host.

Using the above compression optimization will allow us to compress 3 sched_switch events while using only 9 registers: 1 register to store the time delta and 8 to store the event fields. To be able to compress more than 3 events, using a hash would be optimal for storing characters like task name which can be to fit into 32 bits, or use a guest-host shared page to store these characters as debug symbols.

A direct limitation of the event batching mechanism is that we often want to group together the function entry and exit while enabling function tracing, in order to reduce the overhead by 50%. However, functions that last a long time (e.g. close to the root of the call tree such as the main function) would not be collected if tracing was stopped before recording their exit event.

6.1.3 *Architecture*

In this section, we discuss the implementation of our hypertracing technique using hypercall as a transport channel. We have chosen to implement it within the Linux Ftrace infrastructure. Ftrace is an internal Linux kernel tracer, which enables this tracer to investigate problems earlier during the boot-up stage and later during the shutdown stage.

The hypertracing architecture is presented in Fig. 6, where we illustrate the hypergraph and bootlevel internal probing mechanisms for performing hypertracing from a guest system. On the host side, we must enable hypercall tracing to collect hypercalls emitted by either hypergraph or bootlevel; any available Linux tracer can be used for the trace collection.

Hypergraph was developed as a plug-in within Ftrace, which provides the ability to enable function tracing. Unlike the function_graph tracer, hypergraph doesn't need to store collected events into the Ftrace ring buffer. Instead, it directly uses the hypercall API and sends the function address to the host hypervisor. As a result, we can retrieve the VM call stack during any stage, such as boot-up and shutdown, or even during kernel crashes.

A very important feature of this tracer is to associate the call stacks of VM processes with the *vmexits* generated by that VM. This helps to narrow down which function was

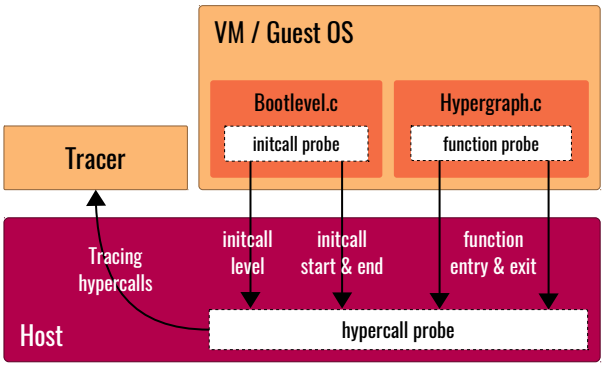

Fig. 6. Hypertracing architecture using hypercalls as transport interface, and illustrating hypergraph and bootlevel tracers.

causing more hypervisor traps.

Bootlevel tracer provides the ability to trace boot levels and built-in modules initializations, known as initcalls. Instead of using the `mcount` hook for tracing initcalls, we used trace-points (static instrumentation) which results in less overhead.

Using hypergraph involves a greater overhead, this is why we advise using the bootlevel tracer at first, in order to get the overall picture, without any noticeable performance degradation. Then, one can use a more fine-grained approach like hypergraph for enabling function tracing and getting in-depth details.

6.1.4 *Performance analysis*

In order to measure the effectiveness of the hypertracing mechanism, by using micro-benchmarks we study the cost of tracing guest kernel events using the traditional tracers such as Lttng, Ftrace and Perf. Then, we compare those tracers against hypertracing with and without batching enabled.

From the results, presented in Fig. 7, we find that batching reduced greatly the hypertracing overhead compared with other tracers. We notice that the overhead is reduced by

a factor of 3 (from 425 ns to 150 ns) with minimal batching (2 events). This performance is even higher than that of the Perf tracer.

The batching cost is a constant overhead of about 25 ns. It involves recording guest timestamps on each event, and computing each event (grouped) time delta to be stored in the payload. Hypertracing outperforms the LTTng and Ftrace tracers when batching 5 events or more together. Thus, with a relatively small number of batched events, it performs better than the finest tracers.

Batching many events may result in having less space for the payload of each event within the hypercall. Batching 15 events results in a 50 ns overhead per event, but in return the hypercall payload will only have 24 bytes available. Therefore, batching 9 events may be better than batching 15 events, because it doubles the payload length to 48 bytes, in exchange for just an additional 12 ns overhead.

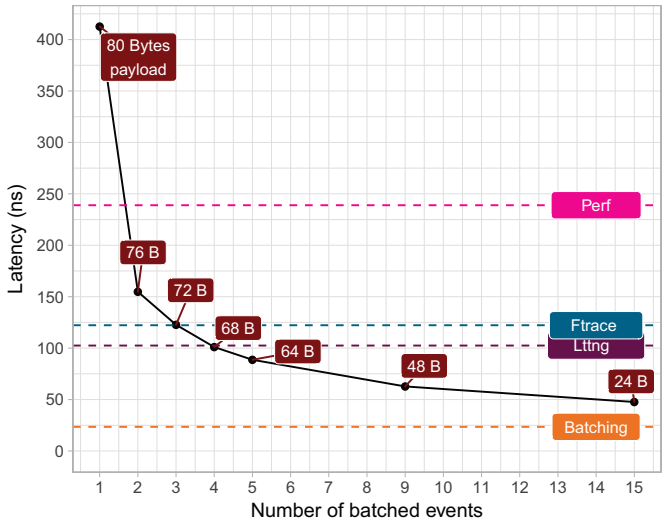

Fig. 7. Offloading latency of the hypertracing, batching is enabled only when at least two events are combined.

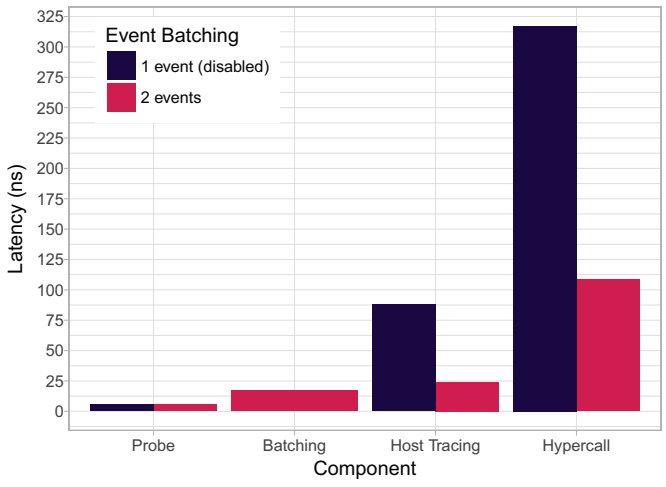

Fig. 8. Different component overhead involved in hypertracing when using event batching.

Tracing through hypercalls involves overhead in different layers. This mechanism starts by hooking a probe callback in the guest trace event infrastructure. The registered callback would be called whenever these events occur. At this point, a hypercall would be triggered using one of the three offloading configurations previously explained. From the host side, the triggered hypercalls are gathered by using a tracer. Any tracer could be used to trace hypercalls, and in this work we are using LTTng due to its low tracing overhead.

In order to understand which part of tracing does impact the performance, we perform another experiment to measure the cost of each component. The results are shown in Fig. 8. By analyzing the results, we find that "batching" not only reduces the hypercall overhead but it does reduce the host tracing overhead as well. As can be seen in the figure, the hypercall is the main source of overhead, being 70% of the overall cost. Host tracing comes second with a 25% cost. On the other hand, using a minimal 2-events batching configuration adds 17 ns of overhead. In return, it reduces the hypercall and host tracing cost by 65% and 72% respectively.

6.2 Hypertracing Through Memory Sharing

Shared-memory based communication channels are known for their high-bandwidth and performance, compared to other existing communication mechanisms. It allows efficient data sharing across virtualization boundaries, by enabling guest-host and inter-guests communication. When communicating with memory, the host hypervisor intervention is only required when performing notification or synchronization. Our primary aim is to prevent any host involvement while designing the trace sharing mechanism. For this reason, we do not use any notification mechanism. In this paper, we mainly focus on communication between guest and host. It is an asynchronous-based communication, where the shared data is consumed without the need for explicit synchronization.

6.2.1 Architecture

We developed the shared memory buffer as a virtual CPU producer-consumer circular buffer. In any buffering design scheme, data overflow may occur when the producer is writing faster than the consumer client reads. In this case, two modes may be used. The first one, is the blocking mode, where the producer suspends writing into the buffer when full, causing new data to be lost. The second, is the flight recorder mode, which overwrites older data in order to push recent data into the buffer.

The flight recorder mode is used here, continuously writing into the shared buffer, to prioritize the recent data. This mode constantly has the latest data, which is convenient when we want to take snapshots of the shared buffer on specific events such as guest exits, task migrations and more.

Fig. 9 shows our shared buffer design. Both the guest and host do write into the shared ring buffer but not at the same time. Guests may use any traditional monitoring tool for recording trace events into the shared buffer. When a trap happens (the hypervisor emulates guest privileged instructions), the host would then record guest related events

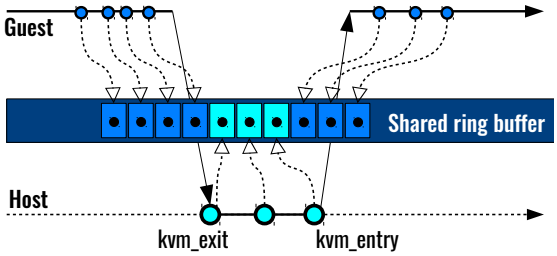

Fig. 9. Shared ring buffer between guest and host.

(VMCS data) into the shared buffer of the preempted virtual CPU. In other words, the host would be responsible for offloading virtualization events to its guests for monitoring purposes, which would then enable guests to be aware of the virtualization operations performed by the hypervisor.

7 Nested Para-virtualization

Nested virtualization is the concept of enabling a VM to run as a hypervisor. VMs running inside guest hypervisors are called nested VMs. Once this feature is enabled by the Infrastructure-as-a-Service (IaaS) provider, cloud users will have the ability to manage and run their favorite hypervisor of choice (like Xen, VMware vSphere ESXi or KVM) as a VM.

The x86 virtualisation is a single-level architecture, it follows the "trap and emulate" model and supports only a single hypervisor mode. Running privileged instructions from a guest level or nested guest level should cause a trap to the host hypervisor (L_0). Any trap received by the host should be inspected; if the trap was coming from a nested level, it should be forwarded to the above hypervisors for emulation. Since the guest hypervisor (L_1) is unmodified, it has the illusion of absolute control of the physical CPU. When L_1 receives the trap forwarded from L_0, L_1 will emulate the nested VM (L_2) trap using VMX instructions. VMX instructions are privileged instructions too, and they can only execute in root mode. In this case, VMX instructions should be emulated by L_0, which causes additional traps [28].

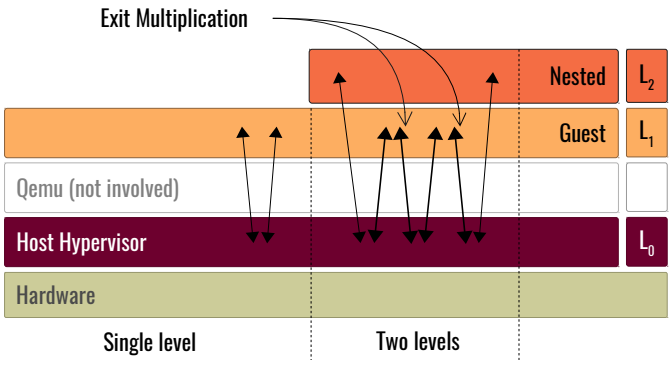

Fig. 10. Exit multiplication, caused by nested traps

This phenomenon is called **exit multiplication**, which is illustrated in Fig. 10, where a single high level L_2 exit causes many L_1 exits leading to performance degradation. The Turtles Project [28] proposed two optimizations implemented in KVM, in order to reduce L_1 exits frequency. The first one optimizes the transitions between L_1 and L_2 by optimizing the merging process of $VMCS_{0\to1}$ and $VMCS_{1\to2}$ into $VMCS_{0\to2}$, copying only the modified fields, and performing multiple fields copy at once. The second optimization relies on improving exit handling in L_1 by replacing *vmread* and *vmwrite* instructions by accessing directly $VMCS_{1\to2}$ by non-trapping memory *load* and *store* operations.

Nested paravirtualization is a way of performing paravirtualization from nested levels, which may result in exit multiplication. If the paravirtualization technique used relies heavily on the hypervisor, it would decrease the guest performance. As presented in section 5, Nahanni (memory sharing region) is the only mechanism that does not require any hypervisor involvement while reading and writing to the shared buffer. This mechanism enables efficient hypertracing through shared memory, where any nested guest may communicate with the host, guest or any nested guest without any performance degradation.

The virtio (page sharing) channel requires notification operations to perform inter-VM communication. This notification is a kick call performed by *pio* operation or a hypercall, either of which will lead to performance slowdown.

Using hypertracing through a hypercall channel may be the most costly mechanism to use from a nested level, because it purely relies on *vmcall*, which is a VMX instruction. To understand the cost involved in nested hypertracing, we performed micro-benchmarks of hypercalls from different layers L_1, L_2 and L_3. The results are presented in Table 1. The table shows that performing a single hypercall from L_1 costs only 286 nanoseconds, whereas performing a hypercall from L_2 (nested guest) costs about 9.3 microseconds, which is about 33 times worse than L_1. And L_3 is the worst, with 232.7 microseconds, about 814 times worse than L_1.

The huge overhead noticed between L_1 and L_2, is the result of exit multiplication happening while the guest hypervisor handles the *vmcall* instruction. For this matter, we propose an optimization in the form of a patch [29] implemented in KVM; it can be ported to other hypervisors as well. This patch optimizes the *vmcall* exit handling in L_0. The concept of this contribution is to avoid involving the nested hypervisor in the communication between the nested guest and the host, which removes any exit multiplication in the nested communication path using hypercalls. Mainly, this prevents the *vmcall* instruction, executed from L_2 (or L_n, with n>1), to be transmitted to L_1, since a single exit L_2 can cause many L_1 exits. This optimization processes *vmcalls* from nested layers with a single exit, for an additional 1-2% overhead compared to L_1, as also shown in table 1.

TABLE 1
VMCALL overhead from nested layers

	Baseline	*Nested vmcall*		*Overhead*	
	L_1	L_2	L_3	L_2	L_3
Default	286 ns	9.3 us	232.7 us	x33	x814
L_0 optimization	286 ns	289 ns	292 ns	1 %	2.1 %

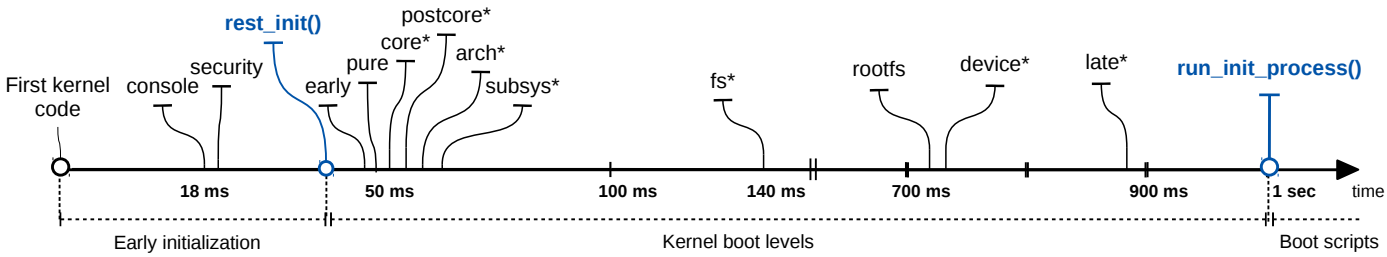

Fig. 11. Linux boot-up sequences.

8 Use Cases

This section shows how we used hypertracing to solve real world issues.

8.1 Early Boot Crashes

Kernel crashes are frequent during the development cycle; kernel developers often use the serial console to inspect debugging messages like printed call stacks or registers values. A more sophisticated and modern way to debug kernel panics is tracing. Tracing enables developers to track back the prior events that led up to the crash. Ftrace implements a feature called ftrace_dump_on_oops. Enabling ftrace_dump_on_oops makes debugging crashes much easier by dumping the entire trace buffer to the console in ASCII format.

Kernel panics happening during the boot-up phase are much more difficult to investigate. The console output is the only available tool during boot-up, but it remains limited when non-trivial issues happen at a very early stage.
Ftrace is an internal Linux kernel tracer; this interesting property makes this tracer most suitable to use for investigating kernel boot issues. We have submitted a kernel patch [30] to the Linux kernel to enable function tracing at a very early stage during boot-up. This contribution enables tracing and debugging during the early kernel boot-up. Moreover, we faced some challenges while developing this feature. We found no way to debug kernel crashes happening during the development of this feature. In early boot-up, the console output is not yet initialized, which makes it almost impossible to debug.

In a virtualized environment, the host system is fully available but does not have access to guest information, and the guest system has all the information but is i
ization phase. Therefore, the goal is to use host c
to debug early guest initialization crashes by us
tualization. Hypercall is the most suitable paravi
technique at that point because it only uses re
vmcall instructions to communicate. At an early
memory sharing or networking cannot be used,
infrastructure-less technique such as the hyper
appropriate for this particular configuration.

We used a combination of both static and dy
instrumentation techniques to insert our tracing
into the kernel code. For dynamic instrumentati
the mcount mechanism that can be enabled using
pilation option called "-pg". We placed a hype
the mcount callback function, which is invoke
function entry. This way, we can get the call stack
to early initialization crashes. With static instrumentation, we instrumented some large functions to dump the values of some variables. We instrumented some inlined functions that were not handled by the mcount mechanism. We also instrumented some assembly code too, (hypercall was a perfect choice for that), because it only needs one instruction (*vmcall*) to perform tracing.

8.2 VM Boot-up

In any cloud platform, the procured VMs need some amount of time to be fully operational for cloud users. Cloud providers require time to select a host in their data centers on which to run the requested new VMs, for resources to be allocated (such as IP addresses) directly to the VM, as well as to copy or boot or even configure the entire OS image. Many surveys and blogs raised questions about these issues. More importantly, cloud users have also complained about this unexpected long provisioning time issue [31]. It hurts their cloud application performance and slows down their development productivity.
In fact, this dynamic process is inevitable to ensure cloud elasticity. A long undesirable latency during this process could result in a degradation of elasticity responsiveness, which will immediately hurt the application performance.

In a cloud system, three distinct stages are performed when a client requests to start a VM. In the first stage, the platform identifies a suitable physical node to allocate the requested VM. The second stage is based on transferring the VM image from the image store to the compute node. While in the final stage, the VM is booting on the physical node [32].
Most cloud providers have a centralized scheduler that orchestrates VMs provisioning. In this matter, the first stage

Boot-up phases

Fig. 12. Boot-up stages.

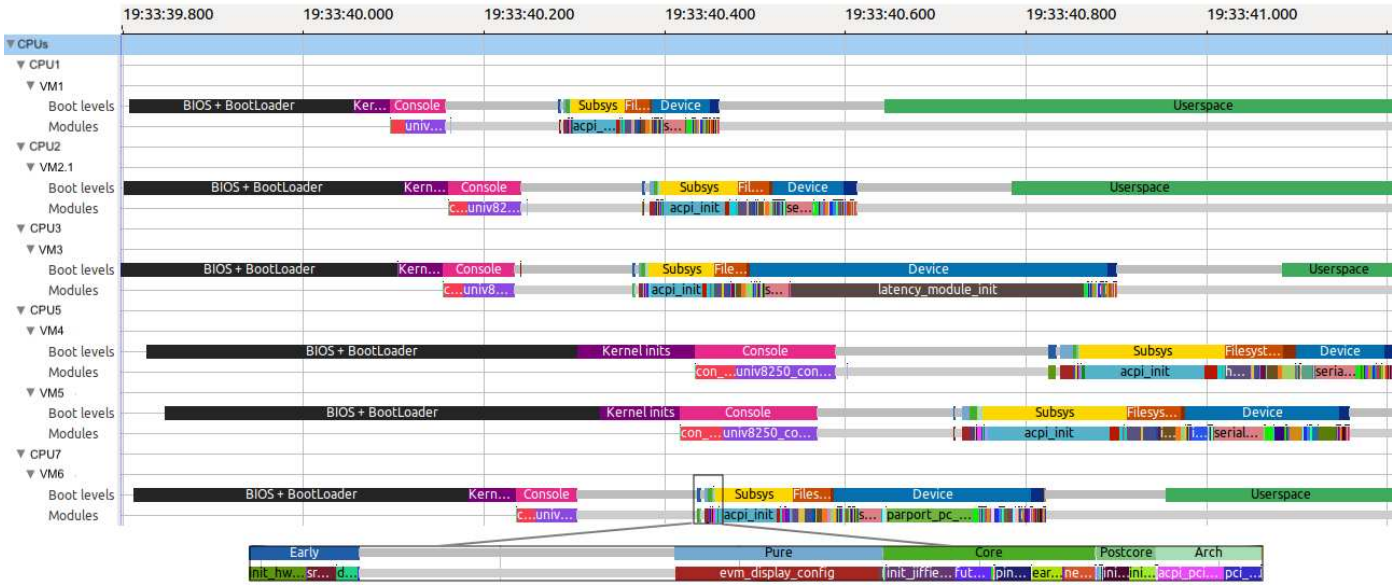

Fig. 13. Tracing VMs boot-up levels and built-in modules.

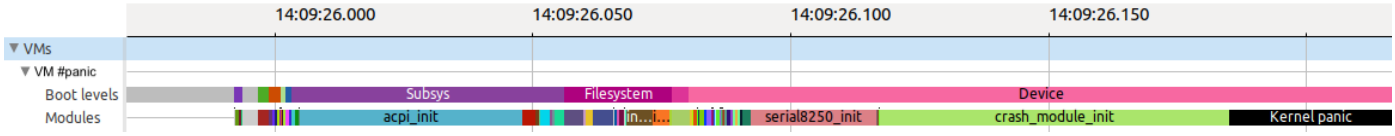

Fig. 14. Detecting kernel panics happening during a VM boot-up.

The VM boot-up stage is performed like any normal Linux boot process, which is illustrated in Fig. 12. When the hypervisor executes the VMLAUNCH instruction, the BIOS starts by performing hardware checks. Then, it locates and loads into the memory the bootloader. Next, the boot loader loads the kernel binary into the main memory and perform a jump to the first kernel code located in *head.S*. Only processor CPU0 is running at this moment, and the kernel performs synchronous initialization with only one thread. Global data structures, CPU, virtual memory, scheduler, interrupt handlers (IRQs), timers, and the console are all initialized in a strict order. This part of the system is behaving like a real-time operating system, and runs fairly quickly. This behavior changes when the kernel runs the function rest_init(); the kernel spawns a new thread to load built-in code (modules) and initialize other processors. The final phase happens when the kernel runs the init process (PID 1), the first userspace program. At this step, the system is preemptible and asynchronous.

Built-in modules are grouped by functionality into separate sections known as boot levels. These sections are defined as follows in a specific order: console, security, early, pure, core, post-core, arch, subsys, fs, rootfs, device and late. Fig. 11 shows the boot sequence order when the kernel boot-up happened in one second.

A built-in module can register itself in any boot level using the kernel initialization mechanism called *initcall* [33]. The boot order of each level is statically defined and identical across all Linux systems. However, the ordering of modules inside each level is determined by the link order. The blacklist feature allows users to prevent some built-in modules from loading during the boot-up, which enables runtime customization of the boot process.

To be able to monitor issues happening during VM boot-ups, we have developed two Ftrace plugins [29]. The first plugin is the **bootlevel** tracer, which enables tracing boot level sequences. The second plugin is the **hypergraph** tracer. This technique provides the ability to trace guest kernel call stacks directly from the host. It works by offloading (through hypercalls) the guest sched_switch and function entry/exit events. This technique can also be used to debug other use cases like unexpected shutdown latencies, or kernel crashes. In this section, hypergraph would be configured to solely trace initcalls (built-in modules) entry & exit, which would help detect latencies during boot-up.

We produced two different experiments to test our method. In both experiments, we enabled host tracing using the following trace events: kvm_hypercall, kvm_exit, kvm_entry and kvm_write_tsc_offset. The first experiment focuses on understanding the boot time in various scenarios where concurrent VMs are booted, whereas the second experiment addresses the case where a crash occurs while loading a specific module.

The result of the first experiment is shown in Fig. 13. In this experiment, we spawned at the same time six VMs VM1, VM2.1 (nested VM within VM2), VM3, VM4, VM5 and VM6, each with one virtual CPU and 2 GB of memory, and pinned their vCPU0 to a dedicated physical processor except for VM4 and VM5, they both have been assigned to a single core (CPU5). The following VMs, VM1, VM2.1, VM3, VM4, and VM5 have identical kernel images. Their kernel

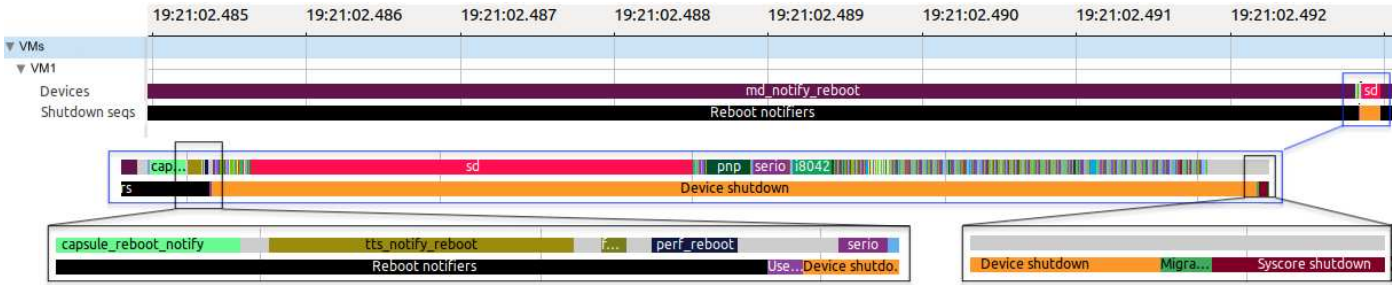

Fig. 15. Tracing VM shutdown sequences.

was configured using the make target *localmodconfig*, which provides a way to disable modules that are not required by the system. In our system, this configuration produced a lightweight kernel image; it contains 830 built-in modules loaded during the boot-up. VM1 and VM2.1 are booting on separate processors CPU1 and CPU2 respectively. But they still have different boot times, about 850 ms and 930 ms respectively. The VM2.1 is booting from nested level (L_2), this explain the overhead (about 9%) compared to VM1.

Next, we study a scenario where two concurrent VMs (VM4 and VM5) are booting on the same CPU. The boot time of VM4 and VM5 is 1.6 and 1.5 seconds respectively, about twice the time it took for VM1 to boot. Furthermore, in Fig. 13 we notice that in either VM4 or VM5, latency was present across all boot sequences. In fact, VM4 and VM5 were preempting each other all the time, and they shared the processor resources equitably through boot sequences.

In VM3, we implemented a custom module named latency_module_init loaded under the device boot level. This module introduces a 324 ms latency at the device level, which added a 35% overhead to the boot-up, and resulting in 1.3 seconds of boot time.

The boot time of VM6 was 1.14 seconds, VM6 was also configured using the make target *localmodconfig*. Additionally, we changed some configuration entries that were setup as ”=m” by ”=y.” When a module is set with ”=m,” it will be loaded as an external module once the boot-up finished. By turning a configuration value from ”=m” to ”=y,” the selected modules will be loaded during the boot-up at the device level. This change added 130 modules to be loaded by VM6 as compared to VM1, and resulted in 27% overhead for the boot-up.

The result of the second experiment is shown in Fig. 14. This experiment aims to verify the effectiveness of hypertracing toward kernel crashes. We implemented a module called crash_module_init; we registered this module to load during device boot level. This module tries to access a non-existing memory which immediately causes a kernel panic. As a result, our monitoring technique detected this crash right away, as seen in Fig. 14.

8.3 VM Shutdown

We verified that the hypertracing method works for tracing late shutdown sequences while rebooting the VM. We used the hypergraph tracer to trace the kernel_restart(char *cmd) function that is related to rebooting the system. The result is shown in Fig. 15.

The function kernel_restart is called after performing a reboot system call from userland. Then, the following sequence is performed:

Reboot notifiers: This step goes through the reboot notifiers list of hooks to be invoked for watchdog tasks. Fig. 15 shows that this step took about 1 second to complete. Most of the time was spent within the md_notify_reboot notifier call. While inspecting the Linux kernel source code, we found that the md_notify_reboot() function effectively performs a call to mdelay(1000) when at least one MD device (Multiple Device Driver or RAID) was used. In our case, the VM was using an MD device which led to the 1 second delays.

Disable usermod: This step ensures that no more userland code will start at this point.

Device shutdown: This step will release all devices on the system, about 340 devices were released in the experiment seen in Fig. 15. Among these devices, the sd device (driver for SCSI disk drives) took about 40% of the total time for this step.

Migrate to reboot cpu: In this step, the kernel forces the scheduler to switch all tasks to one particular CPU. Only a single CPU is running from this point on.

Syscore shutdown: This step performs some shutdown operations like disabling interrupts before powering off the hardware.

8.4 Rolling Upgrade

In a competitive market, cloud and online service providers guarantee the high availability of theirs platform services using Service Level Agreements (SLAs). Availability refers to the time that a service is operating correctly without encountering any downtime, which may be high but cannot reach 100%. For this reason, cloud providers commit to ensure an annual uptime percentage, like Amazon AWS which provides an uptime of at least 99.999%, equivalent to less than 6 minutes per year.

Based on recent studies [34], [35], software upgrades are the major cause of such downtime (planned or unplanned) with up to 50% rates of upgrades failure.

Best practices recommend that industries avoid downtime using **rolling upgrades**. This mechanism enables upgrading and rebooting a single host or domain at a time through the data center [36]. However, this approach may result in temporary performance degradation caused by using the network when transferring the new release to the target nodes, and by having capacity loss while rebooting some nodes. Rolling upgrades are a good solution for

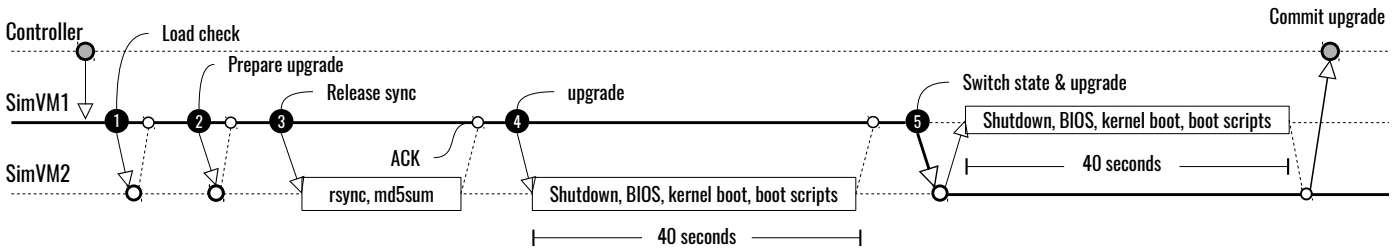

Fig. 16. Performing rolling upgrade on the application simulator.

resources with a high number of identical units, such as nodes in a cloud. There are other resources such as controller nodes or network switches where there are few units and the downtime associated with upgrading each unit is much more problematic, hence the importance in those cases of optimizing the upgrade shutdown and reboot cycle.

With the uprising of the computing as a service model, a big shift was noticed toward hardware simulation, enabling industries to test, simulate and validate their services using simulated hardware at low cost. Using a simulated environment improves development and ensures quality of the embedded software before deploying it to real hardware. A typical fault-tolerant embedded hardware unit is composed of one controller and two (or more) circuit boards defined as primary card and secondary card. The secondary card is used for resiliency purpose; it takes control when the primary card is not available. VMs are used to simulate the hardware unit. A VM is needed to simulate the controller, and two others (SimVM1 and SimVM2) to simulate the primary and secondary cards.

A particularly interesting case we have seen in a product is the presence of some important performance degradation while performing a rolling upgrade within a simulator. An unexpected latency occurred while rebooting the SimVM in the context of an upgrade. Interestingly, this delay was not present when performing the same software upgrade on real hardware. As noted, the difference between the hardware and the simulation is the presence of virtualization and the application simulator.

Fig. 16 illustrates the sequence of the rolling upgrade. The first card to be upgraded is the secondary card SimVM2. The controller asks the primary card (SimVM1) to initiate the upgrade process when the target load (release) was delivered. Then, the steps 1, 2 and 3 are performed by the primary to prepare the secondary for the upgrade. Step 4 involves rebooting the secondary card, and it took about 40 seconds. When the secondary card is booted with the new load, SimVM1 sends a request to SimVM2 to switch its state to primary, then SimVM1 performs the reboot. After step 5, the primary card is now SimVM2, when SimVm1 is booted with the new load it becomes secondary. In the final step, SimVM2 sends to the controller a commit message which indicates that the upgrade was completed.

During the upgrade, the whole system becomes vulnerable and loses its redundancy (fault-tolerance) for about 80 seconds, which is a significant period of time. The boot-up and shutdown sequences are an important part of the upgrade and should be closely investigated.

We traced both the shutdown and boot-up using the hypertracing techniques previously presented (bootlevel and hypergraph tracers). As a result, we found no latency related to the kernel boot-up and the shutdown, they both took about 1 second each to complete. Normally, it is possible to enable monitoring tools after boot-up, but during the upgrade, the stored traces are destroyed, because of the unmounting operations that are performed to prepare for mounting the new release filesystem. To overcome this situation, we decided to use hypertracing to trace boot scripts initialization too. For this matter, we have developed another hypertracing tracer called **hypertrace** [29]. Since we did not know what caused this latency, we enabled some specific events that are presented as follows: scheduling events, system calls, timer and interrupts (IRQs) events.
Analyzing 40 seconds of a trace is very difficult, thus we decided to use the critical path analysis [37]. This analysis is already integrated into TraceCompass[1], and was developed by our research group[2]. The result of the analysis is shown in Fig. 17.

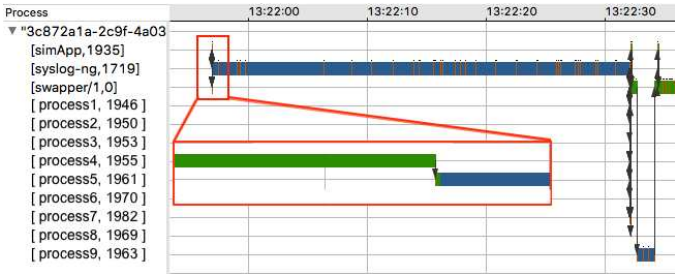

Fig. 17. Active path of the application simulator. Some sensitive information like process's names were hidden, due to privacy concerns.

After the boot-up, the application simulator performs some logging during its initialization; the first logged message was "Starting simulation application" which occurred 30 seconds later than the previous message. When analyzing the results shown in Fig. 17, the active path of the application simulator was blocked by the syslog-ng process for about 30 seconds, this explains the presence of a 30 seconds interval within the simulator logs.

The 30 seconds delay seems like a default configuration for a network timeout. This means that the syslog-ng is somehow offloading logs to an unavailable remote server. However, as we know, syslog would normally use the

1. http://tracecompass.org
2. http://dorsal.polymtl.ca

rsyslog process for remote logging, instead of syslog-ng, and there is no rsyslog process in the trace. While digging into the syslog configuration file, we discovered that syslog was configured for writing over an NFS mount, whereas the remote NFS server was unavailable. When syslog receives the timeout message 30 seconds later, it falls back to the default mode and then uses a local file for logging.

8.5 Virtualization Awareness

We verified that the hypertracing method works through a shared memory region. We used QEMU ivshmem (Nahanni) to enable a shared ring buffer between the guest and host. We enabled guest kernel tracing and configured it to write into the shared buffer. On each hypervisor trap, we record the hypervisor event (VMEXIT and VMENTRY) into the shared buffer. With this configuration, the guest could detect delays related to virtualization.

In this use case, we show how virtualization awareness could help us to find the root cause of latency from guest only. We executed sysbench as a CPU workload (prime numbers) inside the guest which had a single vCPU. Fig. 18 shows the results of guest tracing.

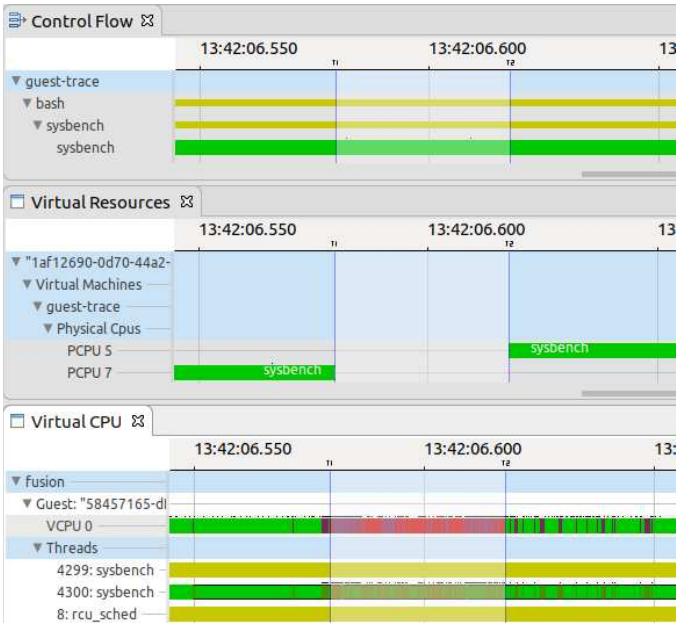

Fig. 18. vCPU migration detection.

We have used three views to explain our findings. The control flow view shows that the sysbench process was running (Green color) on the CPU for the whole period of time. This is the kind of view that we usually get without enabling virtualization awareness. The virtual CPU view shows additional information, such as vCPU preemption states (Purple color) and VM transiting states (Red color) from VMX root mode or non-root. As expected, the vCPU was preempted many times during this experiment, this is a normal behavior. But we notice a long delay of 40 ms caused by a vm_exit with an exit reason of 1. The value 1 means that an external interrupt was triggered from the host to preempt the vCPU.

The virtual resources view shows which physical CPU was used by the guest. We can clearly see that the guest vCPU0 was migrated from pCPU7 to pCPU5. Resulting into a 40 ms latency caused by the vCPU migration.

9 Overhead Analysis

In this section, we compare the overhead of our approach with the existing monitoring tools. Table 3 presents the added overhead of tracing boot-up sequences for existing tracing tools. Our bootlevel tracer achieved a very low overhead of 0.44%; no existing tool can be compared to our tracer because none of them do trace boot levels. Bootgraph and function_graph tracer are both able to trace initcalls, similar to our hypergraph tool. Bootgraph uses the console to print the initcalls duration. Then it parses the dmesg output to retrieve the timings of the initcalls. The Ftrace function_graph tracer uses the mcount mechanism to trace function entries & exits; each event will be recorded to the ring-buffer.

The bootgraph overhead is mostly related to using printk, which will write to the kernel console buffer, an inefficient channel for performance monitoring. The function_graph tracer exhibits the worst performance; this is mainly due to the ftrace initialization time. Ftrace uses memory frame allocation at initialization time to setup it buffers (per-CPU ring-buffer); this avoids postponed page faults when tracing is enabled. However, since we are monitoring the boot-up, the page faults overhead resulting from the frame allocation does significantly impact the performance of the boot time. Hypergraph achieved the lowest overhead at about 0.52%. Unlike bootgraph and function_graph, hypergraph does not rely on ftrace or printk buffers, but instead uses hypercall to offload it events.

In order to get a clear understanding of the overhead involved when using hypertracing, we ran sysbench Disk I/O, CPU and Memory representative workloads to compare our approach with the multi-level [6] tracing approach. We enabled tracepoints that are related to scheduling and system call events inside the VM. We also enabled hypervisor events on the host such as kvm_exit kvm_entry and kvm_hypercall, which are used by all the approaches. Moreover, we enabled synchronization events that were needed for the multi-level approach to work. The results are presented in Table 2.

Hypertracing through shared memory has the minimal overhead among all approaches. Particularly with I/O workloads, our approach incurs the lowest overhead of about 6%. The reason behind this low overhead is that we have configured the shared buffer to be consumed asynchronously from a different host CPU, it means that all I/O operations related to tracing were handled by a different host process running on a dedicated CPU.

Unlike hypertracing, the multi-level approach consumes it trace buffer from inside the VM. Doing I/O operations inside a VM require hypervisor involvement for handling the I/O device virtualization, resulting in a more context switching between guest and hypervisor which induce much greater overhead of about 11%.

Hypertracing through hypercalls shows a significant overhead of 38%, due to the higher presence of sys_write events. Enabling event batching did significantly reduce the overhead from 38% to 25%, about a 70% improvement

TABLE 2
Comparison of multi-level tracing approach [6] with our hypertracing approach for synthetic loads

Benchmark	Baseline	Multi-level tracing	Hypertracing			Overhead (%)			
			Shared memory	Hypercall	Batching	Multi-level	Shared memory	Hypercall	Batching
File I/O (ms)	52.380	58.220	55.960	72.714	65.580	11.15	6.83	38.82	25.2
Memory (ms)	525.788	538.544	537.530	540.368	537.118	2.42	2.23	2.77	2.15
CPU (ms)	1380.34	1428.076	1421.426	1430.085	1426.404	3.45	2.97	3.6	3.33

TABLE 3
Comparison of existing boot-up tracing tools and our hypertracing approach

Boot-up tracing	Time (ms)	Overhead (%)
Baseline	734.43	-
Bootlevel	737.67	0.44
Hypergraph (initcalls)	738.28	0.52
Bootgraph (initcalls)	740.45	0.81
Ftrace function_graph (initcalls)	743.06	1.17

since we only used a minimal batching configuration to group system call entry and exit events. Using a more aggressive batching configuration will definitely reduce the overhead even more. With CPU and memory workloads, all the approaches had similar overheads, around 2% and 3% respectively. Hypertracing performed slightly better compared to multi-level when enabling event batching.

10 Conclusion

In this paper, we presented hypertracing, a paravirtualization technique that we used as new monitoring infrastructure for investigating virtual machines performance issues.

We presented the strong and weak points of existing inter-VM communication channel alternatives to TCP/IP. Then, we compared their throughput and concluded that memory sharing achieved higher efficiency. We then explained how we developed hypertracing using hypercalls, which enables debugging VMs within sensitives phases such as system crashes, boot-up and shutdown. We also showed how hypertracing can be used with the shared-memory-based approach for enabling virtualization awareness within VMs.

Finally, we proposed an L_0 optimization in the form of a patch to KVM. This contribution enables efficient nested paravirtualization, and allows our technique to monitor any nested environment without additional overhead. Current monitoring tools incur a significant overhead due to exit multiplication caused by I/O operations.

In the future, we aim to combine hypercall and memory sharing techniques to enable advanced hypertracing mechanisms. This configuration would combine the advantages of each channel, and thus enable complex issues to be solved.

Acknowledgments

The authors would like to thank NSERC, Prompt, Ericsson, Ciena, Google and EfficiOS for funding this research project as well as for their helpful interactions which made this work possible. We also would like to thank Genevive Bastien for her help with TraceCompass, and Hani Nemati and Mohamad Gebai for their helpful discussions regarding QEMU/KVM.

References

[1] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*. ACM, 2010, pp. 35–46.

[2] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 423–430.

[3] D. Marshall, "Understanding full virtualization, paravirtualization, and hardware assist," *VMWare White Paper*, 2007.

[4] E. Khen, N. J. Zaidenberg, A. Averbuch, and E. Fraimovitch, "Lgdb 2.0: Using lguest for kernel profiling, code coverage and simulation," in *Performance Evaluation of Computer and Telecommunication Systems (SPECTS), 2013 International Symposium on*. IEEE, 2013, pp. 78–85.

[5] D. Stolfa, "Tracing virtual machines in real-time," Master's thesis, University of Rijeka. Faculty of Engineering., August 2017.

[6] M. Gebai, F. Giraldeau, and M. R. Dagenais, "Fine-grained preemption analysis for latency investigation across virtual machines," *Journal of Cloud Computing*, vol. 3, no. 1, p. 23, 2014.

[7] C. Biancheri and M. R. Dagenais, "Fine-grained multilayer virtualized systems analysis," *Journal of Cloud Computing*, vol. 5, no. 1, p. 19, 2016.

[8] Y. Yunomae, "Integrated trace using virtio-trace for a virtualization environment," LinuxCon North America/CloudOpen North America,New Orleans, LA. Keynote presentation, 2013.

[9] S. Rostedt, "Finding origins of latencies using ftrace," in *11th Real-Time Linux Workshop*, 2009, pp. 28–30.

[10] H. Jin, W. Cao, P. Yuan, and X. Xie, "Xenrelay: An efficient data transmitting approach for tracing guest domain," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 258–265.

[11] H. Nemati and M. R. Dagenais, "Virtual cpu state detection and execution flow analysis by host tracing," in *Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on*. IEEE, 2016, pp. 7–14.

[12] H. Nemati, S. D. Sharma, and M. R. Dagenais, "Fine-grained nested virtual machine performance analysis through first level hypervisor tracing," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2017, pp. 84–89.

[13] S. D. Sharma, H. Nemati, G. Bastien, and M. Dagenais, "Low overhead hardware-assisted virtual machine analysis and profiling," in *Globecom Workshops (GC Wkshps), 2016 IEEE*. IEEE, 2016, pp. 1–6.

[14] J. Hwang, S. Zeng, F. y Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*. IEEE, 2013, pp. 269–276.

[15] J. Che, Q. He, Q. Gao, and D. Huang, "Performance measuring and comparing of virtual machine monitors," in *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, vol. 2. IEEE, 2008, pp. 381–386.

[16] V. Goyal, N. Horman, K. Ohmichi, M. Soni, and A. Garg, "Kdump: Smarter, easier, trustier," in *Linux Symposium*, 2007, p. 167.

[17] V. Romanovsky, "Libvirt: Enable panic device notification," January 2018. [Online]. Available: https://blueprints.launchpad.net/nova/+spec/libvirt-enable-pvpanic

[18] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, p. 26, 2018.

[19] J. Corbet, "The perils of printk," November 2016. [Online]. Available: https://lwn.net/Articles/705938/

[20] Y. Ren, L. Liu, Q. Zhang, Q. Wu, J. Wu, J. Kong, J. Guan, and H. Dai, "Residency-aware virtual machine communication optimization: Design choices and techniques," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 823–830.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 164–177.

[22] K. K. Ram, J. R. Santos, and Y. Turner, "Redesigning xens memory sharing mechanism for safe and efficient i/o virtualization," in *Proceedings of the 2nd conference on I/O virtualization*. USENIX Association, 2010, pp. 1–1.

[23] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[24] G. Lettieri, V. Maffione, and L. Rizzo, "A study of i/o performance of virtual machines," *The Computer Journal*, vol. 61, no. 6, pp. 808–831, 2018. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxx092

[25] S. Patni, J. George, P. Lahoti, and J. Abraham, "A zero-copy fast channel for inter-guest and guest-host communication using virtio-serial," in *Next Generation Computing Technologies (NGCT), 2015 1st International Conference on*. IEEE, 2015, pp. 6–9.

[26] A. C. Macdonell *et al.*, *Shared-memory optimizations for virtual machines*. University of Alberta, 2011.

[27] K. T. Raghavendra, "Kvm: Add documentation on hypercalls," Linux Documentation Source Code, 2012. [Online]. Available: linux/ Documentation/virtual/kvm/hypercalls.txt

[28] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization." in *OSDI*, vol. 10, 2010, pp. 423–436.

[29] A. Benbachir, "Online repository," January 2018. [Online]. Available: https://github.com/abenbachir/hypertracing

[30] ——, "ftrace: support very early function tracing," January 2018. [Online]. Available: https://lkml.org/lkml/2017/11/8/1031

[31] T. Hoff, "Are long vm instance spin-up times in the cloud costing you money?" January 2018. [Online]. Available: http://highscalability.com/blog/2011/3/17/are-long-vm-instance-spin-up-times-in-the-cloud-costing-you.html

[32] T. L. Nguyen and A. Lebre, "Virtual machine boot time model," in *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*. IEEE, 2017, pp. 430–437.

[33] M. Gilber, "Kernel initialization mechanisms," June 2005. [Online]. Available: https://lwn.net/Articles/141730/

[34] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., 2009, p. 18.

[35] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in mirage, an integrated software upgrade testing and distribution system," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 221–236, 2007.

[36] E. A. Brewer, "Lessons from giant-scale services," *IEEE Internet Computing*, vol. 5, no. 4, pp. 46–55, 2001.

[37] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, 2016.

Abderrahmane Benbachir finished his BEng and MS degree in Computer Engineering at Polytechnique Montreal. He joined the DORSAL lab to work on cutting-edge virtualization technology. He is now a software developer at Microsoft, his research interests include performance analysis and cloud computing.

Michel Dagenais is a professor at Ecole Polytechnique de Montreal, in the Computer and Software Engineering Department. His research interests include several aspects of multicore distributed systems with emphasis on Linux and open systems. His group has made several original contributions to Linux.